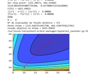
```
F(X1) = 6421.49823
||x^2 - x^1|| / ||x^2|| = 0.00213
|f(x^2) - f(x^1)| / |x^2| = 0.00000
Our stop point: [215.57278, 464.55958]
S1[-0.2788100263717297, -0.15456527020198686]
F(X1) = 6421.49874
||x^2 - x^1|| / ||x^2|| = 0.00066
|f(x^2) - f(x^1)| / |x^2| = 0.00000
Our stop point: [215.54012, 464.48081]
S1[-0.03825415823166595, -0.15877408138617167]
F(X1) = 6421.49844
||x^2 - x^1|| / ||x^2|| = 0.00031
|f(x^2) - f(x^1)| / |x^2| = 0.00000
Our stop point: [215.44268, 464.07149]
S1[-0.18384274036752357, -0.32932077882122965]
F(X1) = 6421.49825
||x^2 - x^1|| / ||x^2|| = 0.00067
|f(x^2) - f(x^1)| / |x^2| = 0.00000
Our stop point: [215.44852, 464.17289]
S1[-0.0025157416797264845, 0.10139858947025004]
F(X1) = 6421.49823
||x^2 - x^1|| / ||x^2|| = 0.00020
|f(x^2) - f(x^1)| / |x^2| = 0.00000
Our stop point: [215.44672, 464.15498]
S1[0.8463935408073259188, -0.01790086271221618988]
F(X1) = 6421.49822
||x^2 - x^1|| / ||x^2|| = 0.00004
|f(x^2) - f(x^1)| / |x^2| = 0.00000
DONE
N = 8
Nº de avaliações da função objetivo = 372
Ponto ótimo = [215.44671541417308, 464.1549793621796]
Função objetivo no ótimo = 6421.49822
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:53: UserWarning: The following kwargs were not used by contour: 'linewidth'
```



2º Método de otimização direto - ***Powell***

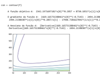
```
# Powell é um método de busca multivariável ao longo de direções conjugadas

# Duas direções d1 e d2 são ditas conjugadas entre si, com respeito a uma matriz H, se:
# d1.T H d2 = 0

# A ortogonalidade é um caso particular de conjugância, onde H = I.

# O método de Powell também permite que a função seja não-diferenciável, já que não usa suas derivadas.

# No entanto, ele usa uma informação adicional em relação ao Rosenbrock, que é o alfa (tamanho do passo na direção d)

# obtido por um método de otimização monovariável, no caso aqui sendo o método da seção áurea ("golden search").
```

ADICIONAL: Método de otimização direto - ***Powell*** por meio do Scipy.optimize.minimize:

```
import numpy as np
from scipy.optimize import minimize

x0 = np.array([100, 100])

# Definindo a função objetivo f(x):

def f(x):
    return 2342.1971697181*x[0]**0.2857 + 8730.1851*(x[1]/x[0])**0.2857 + 62823.6915040394*(1/x[1])**0.2857 - 26190.5553

res = minimize(f, x0, method='powell', options={'ftol': 1e-9, 'disp': True})

res.x

    Optimization terminated successfully.
             Current function value: 6421.498224
             Iterations: 5
             Function evaluations: 175
    /usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:8: RuntimeWarning: invalid value encountered in double_scalars
      if __name__ == '__main__':
    array([215.44346447, 464.158888  ])
```

1º Método de otimização indireto - ***Gradiente (Steepest Descent)***

```
# Primeiro, vamos implementar o método steepest descent em python (completo com visualização gráfica):

# Lembrando que o método steepest descent é um método indireto, ou seja, ele faz uso da derivada de f (df_x).

# Nesse bloco de código, vamos utilizar métodos simbólicos para o cálculo dos gradientes (sympy).

# Além disso, outros métodos como Newton fazem uso da Hessiana explicitamente, então é conveniente calcular aqui também.

import numpy as np
import pandas as pd
from numpy import linalg as LA
import matplotlib.pyplot as plt
import math
import sympy
from sympy.utilities.lambdify import lambdify

# Extraindo a função objetivo f, em termos simbólicos (.sympy.lambdify):

def f_x(f_expression, values):
    f = lambdify((v[0],v[1]), f_expression)
    return f(values[0],values[1])

# Extraindo os gradientes de f:

def df_x(f_expression, values):
    df1_sympy = np.array([sympy.diff(f_expression, i) for i in v])       #derivadas de 1ª ordem
    dfx_0 = lambdify((v[0],v[1]), df1_sympy[0])                          #derivada em relação à x_0
    dfx_1 = lambdify((v[0],v[1]), df1_sympy[1])                          #derivada em relação à  x_1
    evx_0 = dfx_0(values[0], values[1])                                  #calculando o gradiente nos valores dados
    evx_1 = dfx_1(values[0], values[1])
    return(np.array([evx_0,evx_1]))

# Extraindo a Hessiana:

def hessian(f_expression):
    df1_sympy = np.array([sympy.diff(f_expression, i) for i in v])             #derivadas de 1ª ordem
    hessian = np.array([sympy.diff(df1_sympy, i) for i in v])                  #hessiana
    return(hessian)

# Aqui, vamos criar 2 funções para visualizar o caminho que cada método realiza sobre as curvas de níveis da função objetivo.

# Função para criar um plot de curvas de níveis:

def contour(sympy_function):
    x = np.linspace(50, 300, 100)
    y = np.linspace(50, 600, 100)
    x, y = np.meshgrid(x, y)
    func = f_x(sympy_function, np.array([x,y]))
    return plt.contour(x, y, func)

# Função para plotar com o caminho do algoritmo:

def contour_travel(x_array, sympy_function):
    x = np.linspace(50, 300, 100)                         #x-axis
    y = np.linspace(50, 600, 100)                         #y-axis
    x, y = np.meshgrid(x, y)                              #criando o grid x & y
    func = f_x(sympy_function, np.array([x,y]))
    plt.contour(x, y, func)
    plot = plt.plot(x_array[:,0],x_array[:,1],'x-')
    return (plot)

v = sympy.Matrix(sympy.symbols('x[0] x[1]'))

# Criando a função objetivo do exercício (8):

f_sympy8 = 8730.1851*((v[0]/100)**0.2857 + (v[1]/v[0])**0.2857 + (1000/v[1])**0.2857 - 3)

# Alocando-a como "f":

f = f_sympy8

# Extraindo a função:

f = f_x(f, v)
print('A função objetivo é: ',f)
print()

# Encontrando o gradiente da função:

df = df_x(f, v)
print('O gradiente da função é: ',df)
print()

# Encontrando a Hessiana da função:            # A etapa da Hessiana é opcional,
                                               # porque não vamos usá-la explicitamente nos métodos aqui considerados,
hess_f = hessian(f)                            # que fazem aproximações de H.
print('A Hessiana da função é: ',hess_f)
print('')

# Plotando as curvas de níveis:

con = contour(f)

    A função objetivo é:  2342.1971697181*x[0]**0.2857 + 8730.1851*(x[1]/x[0])**0.2857 + 62823.6915040394*(1/x[1])**0.2857 - 26190.5553

    O gradiente da função é:  [669.165731388461*x[0]**(-0.7143) - 2494.21388387*(x[1]/x[0])**0.2857/x[0]
     2494.21388387*(x[1]/x[0])**0.2857/x[1] - 17948.7286627041*(1/x[1])**0.2857/x[1]]

    A Hessiana da função é:  [Derivative(669.165731388461*x[0]**(-0.7143) - 2494.21388387*(x[1]/x[0])**0.2857/x[0], x[0]) ...
```



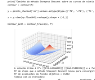
```
# Inicializando o contador externamente

count = 0

# Definindo o contador que entrará junto com a função fun(x), que dirá quantas vezes fun(x) foi avaliada

def incCount():
    global count
    count = count+1

# Definindo a função objetivo fo(x1,x2):

def fo(x1,x2):
    incCount()
    return 2342.1971697181*x1**0.2857 + 8730.1851*(x2/x1)**0.2857 + 62823.6915040394*(1/x2)**0.2857 - 26190.5553


# Definindo o gradiente da função:

def gradf(x1,x2):
    return np.matrix([[669.165731388461*x1**(-0.7143) - 2494.21388387*(x2/x1)**0.2857/x1],
                      [2494.21388387*(x2/x1)**0.2857/x2 - 17948.7286627041*(1/x2)**0.2857/x2]])

# Visualização gráfica da função para certificar que está correta:

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

fig = plt.figure()
ax = fig.gca(projection='3d')

X = np.linspace(150, 300, 50)
Y = np.linspace(350, 500, 50)
X, Y = np.meshgrid(X, Y)
Z = fo(X,Y)

surf = ax.plot_surface(X, Y, Z,cmap=cm.coolwarm)
ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
ax.set_zlabel('$f(x1,x2)$');

plt.show()
```

```
# Definindo o método da seção áurea ("golden search") customizado para minimizar o "alfa" (passo). "[ak, bk]" é o intervalo.

# "length_uncertainty" é a incerteza admitida para a solução. "Y" é o ponto inicial e "d" a direção de busca:

def golden_search(ak, bk,length_uncertainty, Y, d):

    k=1
    termination=True

    # Definindo as direções X1 e X2 e o valor do ponto inicial:

    x1_in=float(Y[0])
    x2_in=float(Y[1])
    d1=float(d[0])
    d2=float(d[1])

    # Inicialização. Calculando alfa, u e os valores das funções:

    gama=((-1+math.sqrt(5))/2)                        #gama é o "nº áureo" (0,618)
    alfak=gama*ak+(1-gama)*bk
    uk=(1-gama)*ak+gama*bk

    # Calculando os valores das funções usando o ponto e a direção dados, e o alfa e u calculados:

    falfak=fo(x1_in+alfak*d1, x2_in+alfak*d2)
    fuk=fo(x1_in+uk*d1,x2_in+uk*d2 )

    while termination:

        # Definindo novos valores para o intervalo:

        if falfak>fuk:
            ak=ak
            bk=uk
            uk=alfak
            alfak=gama*ak+(1-gama)*bk
            fuk=falfak
            falfak=fo(x1_in+alfak*d1, x2_in+alfak*d2)

        else:
            ak=alfak
            bk=bk
            alfak=uk
            uk=(1-gama)*ak+gama*bk
            falfak=fuk
            fuk=fo(x1_in+uk*d1,x2_in+uk*d2 )

        # Checando o critério de parada:

        if (bk-ak)<length_uncertainty:

            # A solução ótima está dentro do intervalo [ak,bk].

            # Escolhendo o ponto do meio como solução ótima:

            search_magnitude=(bk+ak)/2
            termination=False

        else:
            k=k+1
    return search_magnitude
```

```
# Definindo o método do gradiente ("steepest descent")

# Definindo um dataframe (pandas.dataframe) para salvar as soluções e direções para cada iteração:

points_checked=pd.DataFrame(columns=["xk","dk","alfa","x(k+1)","gradiente","fun obj"])

# Escolhendo o ponto inicial, tamanho da incerteza e calculando a primeira direção do gradiente:

len_uncert=1e-9
Xk=np.matrix([[100],[100]])
n=int(Xk.shape[0])
dk=-gradf(Xk[0,0],Xk[1,0])
termination=True

while termination:

    # Calculando a magnitude da direção (alfad) usando o método da seção áurea ("golden search"):

    alfad=golden_search(0,100,1e-9,Xk,dk)

    # Calculando o novo ponto:

    Xk1=Xk+alfad*dk

    # Salvando a informação no dataframe:

    points_checked=points_checked.append({"xk":(np.round(Xk[0,0],2),np.round(Xk[1,0],2)),
                                          "dk":(np.round(dk[0,0],2),np.round(dk[1,0],2)),
                                          "alfa":np.round(alfad,3),
                                          "x(k+1)":np.round(Xk1,2),
                                          "gradiente":np.round(gradf(Xk1[0,0],Xk1[1,0]),2),
                                          "fun obj":np.round(fo(Xk1[0,0],Xk1[1,0]),2)}, ignore_index=True)

    # Atualizando a direção na direção do gradiente "steepest":

    dk=-gradf(Xk1[0,0],Xk1[1,0])

    # Checando o critério de término:

    if LA.norm(gradf(Xk1[0,0],Xk1[1,0]))<len_uncert:
        termination=False
        print("A solução ótima é X*=",Xk1[0],Xk1[1], "e a função objetivo no ótimo é = ",np.round(fo(Xk1[0,0],Xk1[1,0]),2)  )

    #Atualizando Xk:

    Xk=Xk1
```

```
print('Nº de steps que o método Steepest Descent levou para convergir: ', len(points_checked))

print("Nº de avaliações da função objetivo = " + str(count))

print("Tabela com as iterações: ")

print(points_checked)

print("Caminho do método Steepest Descent sobre as curvas de níveis de f:")
contour = contour(f)

y = points_checked['xk'].values.astype(dtype=[('f0', '<f8'), ('f1', '<f8')])       # Essa é uma transformação que será explicada
                                                                                   # no arquivo EXTRA.
z = y.view(np.float64).reshape(y.shape + (-1,))

Contour_path = contour_travel(z, f)
```

```
A solução ótima é X*= [[215.44346899]] [[464.15888336]] e a função objetivo no ótimo é =  6421.49
Nº de steps que o método Steepest Descent levou para convergir:  275
Nº de avaliações da função objetivo = 15682
Tabela com as iterações:
                   xk            dk  ...         gradiente  fun obj
0          (100, 100)  (-0.0, 23.21)  ...   [[-9.71], [0.0]]  6800.34
1      (100.0, 316.23)   (9.71, -0.0)  ...   [[0.0], [-1.66]]  6521.37
2     (177.83, 316.23)   (-0.0, 1.66)  ...  [[-1.42], [-0.0]]  6445.77
3     (177.83, 421.7)    (1.42, 0.0)  ...   [[0.0], [-0.32]]  6427.64
4     (205.35, 421.7)   (-0.0, 0.32)  ...  [[-0.3], [-0.0]]  6423.02
..                ...            ...  ...              ...      ...
270   (215.44, 464.16)   (-0.0, 0.0)  ...   [[-0.0], [0.0]]  6421.49
271   (215.44, 464.16)   (0.0, -0.0)  ...   [[0.0], [-0.0]]  6421.49
272   (215.44, 464.16)   (-0.0, 0.0)  ...   [[-0.0], [0.0]]  6421.49
273   (215.44, 464.16)   (0.0, -0.0)  ...   [[0.0], [-0.0]]  6421.49
274   (215.44, 464.16)   (-0.0, 0.0)  ...   [[-0.0], [0.0]]  6421.49

[275 rows x 6 columns]
Caminho do método Steepest Descent sobre as curvas de níveis de f:
```

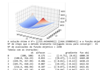


2º Método de otimização indireto - ***Gradiente Conjugado ("Conjugate Gradient: Fletcher-Reeves")***:

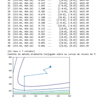
```
# Inicializando o contador externamente

count = 0

# Definindo o contador que entrará junto com a função fun(x), que dirá quantas vezes fun(x) foi avaliada

def incCount():
    global count
    count = count+1

# Definindo a função objetivo fo(x1,x2):

def fo(x1,x2):
    incCount()
    return 2342.1971697181*x1**0.2857 + 8730.1851*(x2/x1)**0.2857 + 62823.6915040394*(1/x2)**0.2857 - 26190.5553


# Definindo o gradiente da função:

def gradf(x1,x2):
    return np.matrix([[669.165731388461*x1**(-0.7143) - 2494.21388387*(x2/x1)**0.2857/x1],
                      [2494.21388387*(x2/x1)**0.2857/x2 - 17948.7286627041*(1/x2)**0.2857/x2]])

# Visualização gráfica da função para certificar que está correta:

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

fig = plt.figure()
ax = fig.gca(projection='3d')

X = np.linspace(150, 300, 50)
Y = np.linspace(350, 500, 50)
X, Y = np.meshgrid(X, Y)
Z = fo(X,Y)

surf = ax.plot_surface(X, Y, Z,cmap=cm.coolwarm)
ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
ax.set_zlabel('$f(x1,x2)$');

plt.show()
```

```
# Definindo o método do gradiente conjugado ("Fletcher Reeves")

# Definindo um dataframe (pandas.dataframe) para salvar as soluções e direções para cada iteração:

points_checked=pd.DataFrame(columns=["xk","alfacon","dk","alfa","x(k+1)","gradiente","fun obj"])

# Escolhendo o ponto inicial, tamanho da incerteza e calculando a primeira direção do gradiente:

len_uncert=1e-9
Xk=np.matrix([[100],[100]])
n=int(Xk.shape[0])
dk=-gradf(Xk[0,0],Xk[1,0])
termination=True

while termination:

    # Calculando a magnitude da direção (alfad) usando o método da seção áurea ("golden search"):

    alfad=golden_search(0,100,1e-9,Xk,dk)

    # Calculando o novo ponto:

    Xk1=Xk+alfad*dk

    # Calculando "alfacon" para obter uma direção conjugada:

    alfacon=(LA.norm(gradf(Xk1[0,0],Xk1[1,0]))**2)/(LA.norm(gradf(Xk[0,0],Xk[1,0]))**2)

    # Salvando a informação no dataframe:

    points_checked=points_checked.append({"xk":(np.round(Xk[0,0],2),np.round(Xk[1,0],2)),
                                          "alfacon":np.round(alfacon,3),
                                          "dk":(np.round(dk[0,0],2),np.round(dk[1,0],2)),
                                          "alfa":np.round(alfad,3),
                                          "x(k+1)":np.round(Xk1,2),
                                          "gradiente":np.round(gradf(Xk1[0,0],Xk1[1,0]),2),
                                          "fun obj":np.round(fo(Xk1[0,0],Xk1[1,0]),2)}, ignore_index=True)

    # Atualizando a direção gradiente:

    dk=-gradf(Xk1[0,0],Xk1[1,0])+alfacon*dk

    # Checando o critério de término:

    if LA.norm(gradf(Xk1[0,0],Xk1[1,0]))<len_uncert:
        termination=False
        print("A solução ótima é X*=",Xk1[0],Xk1[1], "e a função objetivo no ótimo é = ",np.round(fo(Xk1[0,0],Xk1[1,0]),2)  )

    #Atualizando Xk:

    Xk=Xk1
```

```
print('Nº de steps que o método Gradiente Conjugado levou para convergir: ', len(points_checked))

print("Nº de avaliações da função objetivo = " + str(count))

print("Tabela com as iterações: ")

print(points_checked)

print("Caminho do método Gradiente Conjugado sobre as curvas de níveis de f:")
contour = contour(f)

y = points_checked['xk'].values.astype(dtype=[('f0', '<f8'), ('f1', '<f8')])

z = y.view(np.float64).reshape(y.shape + (-1,))

Contour_path = contour_travel(z, f)
```



```
A solução ótima é X*= [[215.44346902]] [[464.15888341]] e a função objetivo no ótimo é =  6421.49
Nº de steps que o método Gradiente Conjugado levou para convergir:  41
Nº de avaliações da função objetivo = 2298
Tabela com as iterações:
                  xk  alfacon  ...          gradiente  fun obj
0         (100, 100)    0.175  ...   [[-9.71], [0.0]]  6800.34
1    (100.0, 316.23)    0.015  ...   [[0.47], [-1.11]]  6469.48
...
40  (215.44, 464.16)    0.046  ...     [[0.0], [0.0]]  6421.49

[41 rows x 7 columns]
Caminho do método Gradiente Conjugado sobre as curvas de níveis de f:
```

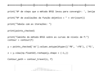


3º Método de otimização indireto - ***BFGS (Broyden-Fletcher-Goldfarb-Shanno)***:

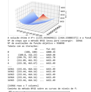
```
# Inicializando o contador externamente

count = 0

# Definindo o contador que entrará junto com a função fun(x), que dirá quantas vezes fun(x) foi avaliada

def incCount():
    global count
    count = count+1

# Definindo a função objetivo fo(x1,x2):

def fo(x1,x2):
    incCount()
    return 2342.1971697181*x1**0.2857 + 8730.1851*(x2/x1)**0.2857 + 62823.6915040394*(1/x2)**0.2857 - 26190.5553


# Definindo o gradiente da função:

def gradf(x1,x2):
    return np.matrix([[669.165731388461*x1**(-0.7143) - 2494.21388387*(x2/x1)**0.2857/x1],
                      [2494.21388387*(x2/x1)**0.2857/x2 - 17948.7286627041*(1/x2)**0.2857/x2]])

# Visualização gráfica da função para certificar que está correta:

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

fig = plt.figure()
ax = fig.gca(projection='3d')

X = np.linspace(150, 300, 50)
Y = np.linspace(350, 500, 50)
X, Y = np.meshgrid(X, Y)
Z = fo(X,Y)

surf = ax.plot_surface(X, Y, Z,cmap=cm.coolwarm)
ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
ax.set_zlabel('$f(x1,x2)$');

plt.show()
```

```
# Definindo o método do BFGS (Broyden-Fletcher-Goldfarb-Shanno)

# Definindo um dataframe (pandas.dataframe) para salvar as soluções e direções para cada iteração:

points_checked=pd.DataFrame(columns=["xk","Dk","dk","alfa","x(k+1)","gradiente","fun obj"])

# Escolhendo o ponto inicial, tamanho da incerteza e calculando a primeira direção do gradiente:

len_uncert=1e-9
Xk=np.matrix([[100],[100]])
n=int(Xk.shape[0])
Dk=np.eye(n)                #Dk = matriz "H(xk)" (inicialmente matriz identidade)
termination=True

while termination:

    #Calculando as direções usando a matriz Dk

    dk=-Dk*gradf(Xk[0,0],Xk[1,0])

    # Calculando a magnitude da direção (alfad) usando o método da seção áurea ("golden search"):

    alfad=golden_search(0,100,1e-9,Xk,dk)

    # Calculando o novo ponto:

    Xk1=Xk+alfad*dk

    # Calculando Ck usando o método BFGS:                         # Ck são aqueles 2 termos de correção de H(xk)

    pk=Xk1-Xk                                                     # pk = "delta x" (Ver notação do slide 57)

    qk=gradf(Xk1[0,0],Xk1[1,0])-gradf(Xk[0,0],Xk[1,0])            # qk = "delta f"

    Ck=(((pk*pk.T)/(pk.T*qk))*float((1+((qk.T*Dk*qk)/(pk.T*qk)))))-((Dk*qk*pk.T+pk*qk.T*Dk)/(pk.T*qk))

        # Salvando a informação no dataframe:

    points_checked=points_checked.append({"xk":(np.round(Xk[0,0],2),np.round(Xk[1,0],2)),
                                          "Dk":np.round(Dk,3),
                                          "dk":(np.round(dk[0,0],2),np.round(dk[1,0],2)),
                                          "alfa":np.round(alfad,3),
                                          "x(k+1)":np.round(Xk1,2),
                                          "gradiente":np.round(gradf(Xk1[0,0],Xk1[1,0]),2),
                                          "fun obj":np.round(fo(Xk1[0,0],Xk1[1,0]),2)}, ignore_index=True)

    # Atualizando Dk

    Dk=Dk+Ck

    # Checando o critério de término:

    if LA.norm(gradf(Xk1[0,0],Xk1[1,0]))<len_uncert:
        termination=False
        print("A solução ótima é X*=",Xk1[0],Xk1[1], "e a função objetivo no ótimo é = ",np.round(fo(Xk1[0,0],Xk1[1,0]),2)  )

    #Atualizando Xk:

    Xk=Xk1
```

```
print('Nº de steps que o método BFGS levou para convergir: ', len(points_checked))

print("Nº de avaliações da função objetivo = " + str(count))

print("Tabela com as iterações: ")

print(points_checked)

print("Caminho do método BFGS sobre as curvas de níveis de f:")
contour = contour(f)

y = points_checked['xk'].values.astype(dtype=[('f0', '<f8'), ('f1', '<f8')])

z = y.view(np.float64).reshape(y.shape + (-1,))

Contour_path = contour_travel(z, f)
```



```
A solução ótima é X*= [[215.44346902]] [[464.15888337]] e a função objetivo no ótimo é =  6421.49
Nº de steps que o método BFGS levou para convergir:  16966
Nº de avaliações da função objetivo = 950098
Tabela com as iterações:
                     xk  ...  fun obj
0            (100, 100)  ...  6800.34
1       (100.0, 316.23)  ...  6469.48
2      (199.79, 357.99)  ...  6421.78
3      (211.09, 461.35)  ...  6421.49
4      (215.38, 463.67)  ...  6421.49
...                 ...  ...      ...
16961  (215.44, 464.16)  ...  6421.49
16962  (215.44, 464.16)  ...  6421.49
16963  (215.44, 464.16)  ...  6421.49
16964  (215.44, 464.16)  ...  6421.49
16965  (215.44, 464.16)  ...  6421.49

[16966 rows x 7 columns]
Caminho do método BFGS sobre as curvas de níveis de f:
```



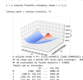
4º Método de otimização indireto - ***DFP (Davidon-Fletcher-Powell)***:

```
# Inicializando o contador externamente

count = 0

# Definindo o contador que entrará junto com a função fun(x), que dirá quantas vezes fun(x) foi avaliada

def incCount():
    global count
    count = count+1

# Definindo a função objetivo fo(x1,x2):

def fo(x1,x2):
    incCount()
    return 2342.1971697181*x1**0.2857 + 8730.1851*(x2/x1)**0.2857 + 62823.6915040394*(1/x2)**0.2857 - 26190.5553


# Definindo o gradiente da função:

def gradf(x1,x2):
    return np.matrix([[669.165731388461*x1**(-0.7143) - 2494.21388387*(x2/x1)**0.2857/x1],
                      [2494.21388387*(x2/x1)**0.2857/x2 - 17948.7286627041*(1/x2)**0.2857/x2]])

# Visualização gráfica da função para certificar que está correta:

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

fig = plt.figure()
ax = fig.gca(projection='3d')

X = np.linspace(150, 300, 50)
Y = np.linspace(350, 500, 50)
X, Y = np.meshgrid(X, Y)
Z = fo(X,Y)

surf = ax.plot_surface(X, Y, Z,cmap=cm.coolwarm)
ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
ax.set_zlabel('$f(x1,x2)$');

plt.show()
```

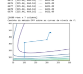
```
# Definindo o método do DFP (Davidon-Fletcher-Powell)

# Definindo um dataframe (pandas.dataframe) para salvar as soluções e direções para cada iteração:

points_checked=pd.DataFrame(columns=["xk","Dk","dk","alfa","x(k+1)","gradiente","fun obj"])

# Escolhendo o ponto inicial, tamanho da incerteza e calculando a primeira direção do gradiente:

len_uncert=1e-9
Xk=np.matrix([[100],[100]])
n=int(Xk.shape[0])
Dk=np.eye(n)                #Dk = matriz "W(xk)" (inicialmente matriz identidade)
termination=True

while termination:

    #Calculando as direções usando a matriz Dk

    dk=-Dk*gradf(Xk[0,0],Xk[1,0])

    # Calculando a magnitude da direção (alfad) usando o método da seção áurea ("golden search"):

    alfad=golden_search(0,100,1e-9,Xk,dk)

    # Calculando o novo ponto:

    Xk1=Xk+alfad*dk

    #Calculando Ck usando o método DFP:                          # Ck são aqueles 2 termos de correção de W(xk)

    pk=Xk1-Xk                                                    # pk = delta x (Ver notação do slide 58)
    qk=gradf(Xk1[0,0],Xk1[1,0])-gradf(Xk[0,0],Xk[1,0])           # qk = delta f
    Ck=((pk*pk.T)/(pk.T*qk))-((Dk*qk*qk.T*Dk)/(qk.T*Dk*qk))

        # Salvando a informação no dataframe:

    points_checked=points_checked.append({"xk":(np.round(Xk[0,0],2),np.round(Xk[1,0],2)),
                                          "Dk":np.round(Dk,3),
                                          "dk":(np.round(dk[0,0],2),np.round(dk[1,0],2)),
                                          "alfa":np.round(alfad,3),
                                          "x(k+1)":np.round(Xk1,2),
                                          "gradiente":np.round(gradf(Xk1[0,0],Xk1[1,0]),2),
                                          "fun obj":np.round(fo(Xk1[0,0],Xk1[1,0]),2)}, ignore_index=True)

    # Atualizando Dk

    Dk=Dk+Ck

    # Checando o critério de término:

    if LA.norm(gradf(Xk1[0,0],Xk1[1,0]))<len_uncert:
        termination=False
        print("A solução ótima é X*=",Xk1[0],Xk1[1], "e a função objetivo no ótimo é = ",np.round(fo(Xk1[0,0],Xk1[1,0]),2)  )

    #Atualizando Xk:
```

```
    Xk=Xk1
```

```
print("Nº de steps que o método DFP levou para convergir: ", len(points_checked))

print("Nº de avaliações da função objetivo = " + str(count))

print("Tabela com as iterações: ")

print(points_checked)

print("Caminho do método DFP sobre as curvas de níveis de f:")
contour = contour(f)

y = points_checked['xk'].values.astype(dtype=[('f0', '<f8'), ('f1', '<f8')])

z = y.view(np.float64).reshape(y.shape + (-1,))

Contour_path = contour_travel(z, f)
```



```
A solução ótima é X*= [[215.443469]] [[464.15888336]] e a função objetivo no ótimo é =  6421.49
Nº de steps que o método DFP levou para convergir:  6680
Nº de avaliações da função objetivo = 374082
Tabela com as iterações:
                  xk  ...  fun obj
0         (100, 100)  ...  6800.34
1      (100.0, 316.23)  ...  6469.48
2     (199.79, 357.99)  ...  6421.78
3     (211.09, 461.35)  ...  6421.49
4     (215.38, 463.67)  ...  6421.49
...              ...  ...      ...
6675  (215.44, 464.16)  ...  6421.49
6676  (215.44, 464.16)  ...  6421.49
6677  (215.44, 464.16)  ...  6421.49
6678  (215.44, 464.16)  ...  6421.49
6679  (215.44, 464.16)  ...  6421.49

[6680 rows x 7 columns]
Caminho do método DFP sobre as curvas de níveis de f:
```



ADICIONAL: Método de otimização indireto - ***BFGS (Broyden-Fletcher-Goldfarb-Shanno)*** por meio do Scipy.optimize.minimize:

```
import numpy as np
from scipy.optimize import minimize

x0 = np.array([100, 100])

# Definindo a função objetivo f(x):

def f(x):
    return 2342.1971697181*x[0]**0.2857 + 8730.1851*(x[1]/x[0])**0.2857 + 62823.6915040394*(1/x[1])**0.2857 - 26190.5553


# Definindo o gradiente da função:

def gradf(x):
    return np.array([669.165731388461*x[0]**(-0.7143) - 2494.21388307*(x[1]/x[0])**0.2857/x[0],
                     2494.21388307*(x[1]/x[0])**0.2857/x[1] - 17948.7286627041*(1/x[1])**0.2857/x[1]])

res = minimize(f, x0, method='BFGS', jac=gradf, options={'gtol': 1e-9, 'disp': True})

res.x
```

```
Optimization terminated successfully.
         Current function value: 6421.490224
         Iterations: 18
         Function evaluations: 21
         Gradient evaluations: 21
array([215.443469  , 464.15888336])
```

Resposta: (b) Podemos fazer comparações entre os métodos, de acordo com a seguinte tabela:

Métodos	Ponto ótimo: x(p2,p3)	Nº de iterações	Nº de avaliações da função objetivo	Função objetivo no ótimo
Rosenbrock	(215,43; 464,15)	8	372	6421,49
Powell (scipy.minimize)	(215,44; 464,16)	5	175	6421,49
Gradiente	(215,44; 464,16)	275	15402	6421,49
Gradiente conjugado	(215,44; 464,16)	41	2296	6421,49
DFP	(215,44; 464,16)	16966	950096	6421,49
BFGS	(215,44; 464,16)	6680	374082	6421,49
BFGS (scipy.minimize)	(215,44; 464,16)	18	21	6421,49

Como podemos verificar, praticamente todos os métodos encontraram o mesmo ponto ótimo, com o mesmo valor da função objetivo. O único método que tem uma ligeira alteração no ponto ótimo é o de Rosenbrock (~0,01 em p2, p3).

A convergência dos métodos foi parecida, porque a função objetivo no caso é bem definida, diferenciável em todo o intervalo de x, e convexa. Se as derivadas de f(x) não estivessem disponíveis, não poderíamos usar os métodos indiretos (em laranja).

A maior diferença que podemos detectar para este caso, é a tendência de um maior número de iterações/avaliações da função objetivo para os métodos indiretos; especialmente para os métodos do DFP e BFGS implementados.

9) Resolva novamente os exercícios (4b) e (4d) utilizando o método SQP e levando-se em conta as restrições do problema e $L(x) \geq 0$.

Solução:

Quando resolvemos os problemas $4(b)$ e $4(d)$, utilizamos o método de otimização **SLSQP**. Na realidade, o SLSQP é análogo ao **SQP**.

Mais precisamente, SQP e SLSQP apresentam o mesmo subproblema da Programação Quadrática (**QP**) em cada etapa do algoritmo.

No SQP, o problema de QP é resolvido por métodos de Programação Quadrática.

Em SLSQP, para resolver o problema de QP, você deve fatorar em LDL^{-1} o lagrangeano da Hessiana e então resolver um problema de mínimos quadrados lineares:

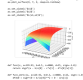
> $\min_{d \in \Re^n} \left\| (D^k)^{1/2} (L^k)^T d + (D^k)^{-1/2} (L^k)^{-1} \nabla f(x^k) \right\|$
> $\nabla g_j(x^k) d + g_j(x^k) = 0, \qquad j = 1, \ldots, m_e,$
> $\nabla g_j(x^k) d + g_j(x^k) \geq 0, \qquad j = m_e + 1, \ldots, m$

Como os dois métodos resolvem o mesmo problema QP, vamos continuar considerando o SLSQP aqui.

E, retomando aos problemas $4(b)$ e $4(d)$, já havíamos considerado as restrições dadas pelo problema original.

A única restrição que está faltando então é $L(x) \geq 0$:

```
#Retornando ao problema 4(b):

import numpy as np
import scipy.integrate
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.optimize import minimize

#Verificando a forma da superfície:

from matplotlib import cm

x1 = np.linspace(1E-3, 2E-2, 50)
x2 = np.linspace(1E-3, 2E-2, 50)
X, Y = np.meshgrid(x1, x2)

Z = 120.93 - 0.5/X - 4000*Y - 25*(X/Y)

fig, ax = plt.subplots(subplot_kw={'projection': '3d'})
ax.plot_surface(X, Y, Z, cmap=cm.rainbow)

ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
ax.set_zlabel('$L(x1,x2)$');
```



```
def func(x, a=120.93, b=0.5, c=4000, d=25, sign=-1.0):
    return sign*(a - b/x[0] - c*x[1] - d*x[0]/x[1])

def func_deriv(x, a=120.93, b=0.5, c=4000, d=25, sign=-1.0):
    dfdx0 = sign*(0 + b/(x[0])**2 + 0 - d/x[1])
    dfdx1 = sign*(0 + 0 - c + d*x[0]/(x[1])**2)
    return np.array([ dfdx0, dfdx1 ])

from scipy.optimize import Bounds
bounds = Bounds([1E-9, 1E-9], [1.0, 1.0])

#Agora, além das restrições de desigualdade x1 <= 0,02, x2 <= x1;
#temos que adicionar também a desigualdade func(x) >= 0:

ineq_cons = {'type': 'ineq',
             'fun' : lambda x: np.array([0.02 - x[0],
                                         x[0] - x[1],
                                         func(x, sign=1.0)]),
             'jac' : lambda x: np.array([[-1.0, 0.0],
                                         [1.0, -1.0],
                                         func_deriv(x, sign=1.0)])}

#Executando a otimização com restrições (constraints = ineq_cons):

x0 = np.array([0.5, 0.5])
res = minimize(func, x0, jac=func_deriv, constraints=ineq_cons,
               method='SLSQP', options={'ftol': 1e-9, 'disp': True},
               bounds = bounds)
print(res.x)
```

```
Optimization terminated successfully.    (Exit mode 0)
            Current function value: -19.409055040739943
            Iterations: 14
            Function evaluations: 21
            Gradient evaluations: 14
[0.01357208 0.00921008]
```

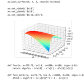
```
#Resultado em notação científica:

scientific_notation1 = "{:.2e}".format(res.x[0])

scientific_notation2 = "{:.2e}".format(res.x[1])

print(scientific_notation1 + "," + scientific_notation2)
```

```
1.36e-02,9.21e-03
```

```
#Valor da função objetivo (lucro) no ponto máximo:

#Lembrar de alterar o sinal novamente para sign=1.0:

def func(x, a=120.93, b=0.5, c=4000, d=25, sign=1.0):
    return sign*(a - b/x[0] - c*x[1] - d*x[0]/x[1])

result = func([1.36E-2, 9.21E-3])

print(result)
```

```
19.40889889506292
```

```
#Retornando ao problema 4(d):

import numpy as np
import scipy.integrate
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.optimize import minimize

#Verificando a forma da superfície:

from matplotlib import cm

x1 = np.linspace(1E-2, 5E-2, 50)
x2 = np.linspace(1E-2, 1E-1, 50)
X, Y = np.meshgrid(x1, x2)

Z = 279.72 - 2.0/X - 4000*Y - 100*(X/Y)

fig, ax = plt.subplots(subplot_kw={'projection': '3d'})
ax.plot_surface(X, Y, Z, cmap=cm.rainbow)

ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
ax.set_zlabel('$L(x1,x2)$');
```



```
def func(x, a=279.72, b=2.0, c=4000, d=100, sign=-1.0):
    return sign*(a - b/x[0] - c*x[1] - d*x[0]/x[1])

def func_deriv(x, a=279.72, b=2.0, c=4000, d=100, sign=-1.0):
    dfdx0 = sign*(0 + b/(x[0])**2 + 0 - d/x[1])
    dfdx1 = sign*(0 + 0 - c + d*x[0]/(x[1])**2)
    return np.array([ dfdx0, dfdx1 ])

from scipy.optimize import Bounds
bounds = Bounds([1E-9, 1E-9], [1.0, 1.0])

#Agora, além das restrições de desigualdade x1 <= 0,02, x2 <= x1;
#temos que adicionar também a desigualdade func(x) >= 0:

ineq_cons = {'type': 'ineq',
             'fun' : lambda x: np.array([0.02 - x[0],
                                         x[0] - x[1],
                                         func(x, sign=1.0)]),
             'jac' : lambda x: np.array([[-1.0, 0.0],
                                         [1.0, -1.0],
                                         func_deriv(x, sign=1.0)])}

#E a otimização com restrições (OPCIONAL - constraints = ineq_cons):

x0 = np.array([0.5, 0.5])
res = minimize(func, x0, jac=func_deriv, constraints=ineq_cons,
               method='SLSQP', options={'ftol': 1e-9, 'disp': True},
               bounds = bounds)
print(res.x)
```

```
Inequality constraints incompatible    (Exit mode 4)
            Current function value: 0.2642368615341253
            Iterations: 11
            Function evaluations: 33
            Gradient evaluations: 10
[0.02001582 0.02001583]
```

```
#Valor da função objetivo (lucro) no ponto máximo (com restrições):

#Lembrar de alterar o sinal novamente para sign=1.0:

def func(x, a=279.72, b=2.0, c=4000, d=100, sign=1.0):
    return sign*(a - b/x[0] - c*x[1] - d*x[0]/x[1])

result = func([0.02001582, 0.02001583])

print(result)
```

```
-0.26423255819140934
```

Resposta: (9) A nova restrição $L(x) \geq 0$ efetivamente não teve nenhum impacto sobre a solução do problema **4(b)**. O novo ponto ótimo e o valor da função objetivo (Lucro) permaneceram inalterados:

> $x = (1,36 \cdot 10^{-2}, 9,21 \cdot 10^{-3})$
> $L(x) = 19,41$

Obviamente, não esperava-se um resultado diferente, visto que a solução do problema 4(b) já satisfazia a nova restrição imposta $L(x) \geq 0$.

Já para a solução do problema **4(d)**, essa nova restrição proporcionou uma leve alteração no resultado, praticamente imperceptível:

> $x = (2,002 \cdot 10^{-2}, 2,002 \cdot 10^{-2})$
> $L(x) = -0,26$

A solução que tínhamos obtido no passado, no problema 4(d), era:

> $x = (2,00 \cdot 10^{-2}, 2,00 \cdot 10^{-2})$
> $L(x) = -0,28$

Esse não era o resultado que se esperava obter, já que essa nova condição imposta gerou uma solução que não satisfaz 2 restrições: $x_1 \leq 0,$ e a própria restrição $L(x) \geq 0$.

Sendo mais criterioso, ela também viola a terceira condição: $x_2 \leq x_1$, na última casa decimal: $x = [0.02001582, 0.02001583]$.

Ou seja, essa nova restrição fez com que todas as outras também fossem violadas.

Esse é um resultado estranho, pois esperava-se simplesmente que o algoritmo não convergisse, já que não existe nenhuma solução possível que atenda às 3 restrições impostas. No entanto, ele convergiu para uma solução que viola as 3 condições ao mesmo tempo.

O algoritmo até indica que reconheceu problemas nesse caso, com a mensagem no início de seu output:

> *Inequality constraints incompatible (Exit mode 4)*

Mas mesmo assim, ele ainda fornece uma resposta final. Provavelmente porque deve existir alguma função dentro do próprio algoritmo do "minimize" que converge para a solução mais próxima desconsiderando-se essa última restrição imposta.